Byte Ordering Concept - Little Endian & Big Endian

Byte ordering in memory refers to the way in which the bytes corresponding to a certain value is stored in the memory. There are two ways in which bytes are stored in memory :

1. Big endian
2. Little endian

Lets understand them one by one :

Big endian

When high order byte or Most significant bit (MSB) is stored at the starting address in memory then this type of systems are known as big endian systems.

Lets consider an example of value 0x0102. If the system is big endian, then the storage in the system would be something like :

Code:

```
increasing Addresses
<--------------------
             02   01
```

As you see above, in the storage, the high order byte (MSB) '01' is stored at the starting or lower address and the lower order byte (LSB) is stored at a higher address.

Little endian

When lower order byte or Least significant bit (LSB) is stored at the starting address in memory then this type of systems are known as little endian systems.

Lets consider an example of value 0x0102. If the system is little endian, then the storage in the system would be something like :

Code:

```
increasing Addresses
<--------------------
             01   02
```

As you see above, in the storage, the Lower order byte (LSB) '02' is stored at the starting or lower address and the high order byte (MSB) is stored at a higher address.

Example

Unfortunately, there is no standard between these two byte orderings and we encounter systems that use both formats. We refer to the byte ordering used by a given system as the host byte order. Lets try to understand the concept through this example :

Code:

```
#include<stdio.h>
#include<stdlib.h>

union{
       short   s;
       char    c[sizeof(short)];
     } un;


int main(int argc, char **argv)
 {
     un.s = 0x0102;

     if (sizeof(short) == 2) {
         if (un.c[0] == 1 && un.c[1] == 2)
             printf("big-endian\n");
         else if (un.c[0] == 2 && un.c[1] == 1)
             printf("little-endian\n");
         else
             printf("unknown\n");
     } else
         printf("sizeof(short) = %lu\n", sizeof(short));
     return 0;
 }
```

- In the above program, we take a value 0x0102. We declared a union with a two byte short 's' and a character array of '2' bytes so that we can reference the same value 0x0102 with both.
[LIST[*]We initialize a union object with this value 0x0102.[*]Now, we check that which byte is kept at lower address and which is kept at higher address using the two bytes of character array. If higher order byte is at lower address then we designate the system as big endian, else its a little endian system.[/LIST]Following is the output at my machine :

Code:

```
 $ ./byteorder
little-endian
```

Byte order functions

Mostly, the data sent on network like IP addresses, ports etc are sent in network byte order or big endian order. So one needs to convert to and from these two byte orders. We have some functions to carry out this task.

From now, we will use network byte order for the byte order of network and host byte order for byte order at local system.

Here is a list of functions :

Code:

```
uint16_t htons(uint16_t host16bitvalue) ;
uint32 t htonl(uint32 t host32bitvalue) ;
        Both return: value in network byte order

uint16_t ntohs(uint16_t net16bitvalue) ;
uint32_t ntohl(uint32_t net32bitvalue) ;
        Both return: value in host byte order
```

- These all functions are found in #include <netinet/in.h>.

- In the names of these functions, 'h' stands for host and 'n' stands for the network.
- Function htons() converts a short (2 byte) value from host byte order to network byte order, While function htonl() converts a long (2 byte) value from host byte order to network byte order.
- Similarly for htonl and ntohl. Its just that 'l' here stands for long value.
- We can think of s as a 16-bit value (such as a TCP or UDP port number) and l as a 32-bit value (such as an IPv4 address).
- When using these functions, we do not care about the actual values (big-endian or little-endian) for the host byte order and the network byte order. What we must do is call the appropriate function to convert a given value between the host and network byte order. On those systems that have the same byte ordering as the Internet protocols (big-endian), these four functions are usually defined as null macros.

Conclusion

To conclude, We should deal with these differences in byte ordering as network programmers because networking protocols must specify a network byte order. For example, in a TCP segment, there is a 16-bit port number and a 32-bit IPv4 address. The sending protocol stack and the receiving protocol stack must agree on the order in which the bytes of these multi byte fields will be transmitted. The Internet protocols use big-endian byte ordering for these multi byte integers. Hence we should be aware of the above fact and use the byte ordering functions whenever and where ever required.

Source: http://www.go4expert.com/articles/byte-concept-little-endian-endian-t26834/